CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Lecture 16: Course Summary and Review

Gregory Gay

DIT636/DAT560 - March 6, 2024

The Impending Exam

- Wednesday, March 13, 8:30 - 12:30
- Practice exam on Canvas.
  - **Somewhat longer than the real exam!**
  - Try solving first without using the sample solutions. Compare your answers.
- Ask questions about any course content!

Topics

- **Quality Attributes**
- **Scenarios**
- **System Testing**
  - **Category Partition**
  - Combinatorial Interaction Testing
- **Exploratory Testing**
- **Unit Testing**
- **Structural Testing**
  - **Control-Flow**
  - **Data-Flow**
- **Mutation Testing**
- Automated Test Generation
- Model-Based Testing
- **Finite State Verification**

Practice Exam

Question 1

1. A program may be reliable, yet not robust.
   a. **True**
   b. False
2. If a system is on an average down for a total 30 minutes during any 24-hour period:
   a. **Its availability is about 98% (approximated to the nearest integer)**
   b. Its reliability is about 98% (approximated to the nearest integer)
   c. Its mean time between failures is 23.5 hours
   d. Its maintenance window is 30 minutes

Question 1

3. A typical distribution of test types is 40% unit tests, 40% system tests, and 20% GUI/exploratory tests.
   a. True
   b. False
4. If a temporal property holds for a finite-state model of a system, it holds for any implementation that conforms to the model.
   a. True
   b. False

Question 1

5. A test suite that meets a stronger coverage criterion will find any defects that are detected by any test suite that meets only a weaker coverage criterion
    - **True**
    - False
6. A test suite that is known to achieve Modified Condition/Decision Coverage (MC/DC) for a given program, when executed, will exercise, at least once:
    - **Every statement in the program.**
    - **Every branch in the program.**
    - Every combination of condition values in every decision.
    - Every path in the program.

Question 1

7. The Category Partition Testing technique requires identification of:
   - **Choices**
   - **Representative values**
   - Def-Use pairs
   - Pairwise combinations
8. Validation activities can only be performed once the complete system has been built.
   - True or **False**
9. Statement coverage criterion never requires as many test cases to satisfy as branch coverage criterion.
   - True or **False**

Question 1

10. Requirement specifications are not needed for selecting inputs to satisfy structural coverage of program code.
    - **True** or False
11. Any program that has passed all test cases and has been released to the public is considered which of the following:
    - Correct with respect to its specification.
    - Safe to operate.
    - Robust in the presence of exceptional conditions.
    - **Considered to have passed verification.**

Question 2

Consider the software for air-traffic control at an airport.

Identify one performance, one availability, and one security requirement that you think would be necessary for this software and develop a quality scenario for each.

Question 2

Performance Requirement: Under normal load (< 500 aircraft), displayed aircraft positions shall be updated on a user's display in under 55 ms.

Performance Scenario:

- **Overview:** Check system responsiveness for displaying aircraft positions
- **System state:** Deployment environment working correctly with less than 500 tracked aircraft.
- **Environment state:** All aircraft tracking hardware is functional.
- **External stimulus:** 50 Hz update of ATC system.
- **System response:** radar/sensor values are computed, new position is displayed to the air traffic controller with maximum error of 5 meters.
- **Response measure:** Fusion and display process completes in less than 45 ms 95% of the time, and in less than 50 ms 99% of the time. There is an absolute deadline of 55 ms.

Question 2

Availability Requirement: The system shall be able to tolerate the failure of any single server host, graphics card, display or network link.

Availability Scenario:

- Overview: One of the monitor display cards fails during transmission of a screen refresh.
- System State: System is working correctly under normal load with no failures.
- Environment state: No relevant environment factors.
- External stimulus: A display card fails.
- Required system response: failure detected within 10 ms and display information routed through redundant graphics card with no user-discernable change to display. Graphics card failure will be displayed as error message at bottom right hand of ATC display.
- Response measure: no loss in continuity of visual display and failover with visual warning completes within 1 s.

Question 2

Security Requirement: The system shall maintain audit logs of any logins to the ATC database, containing sufficient information to identify an attacker.

Security Scenario:

- Overview: A malicious agent gains access to the flight records database in the ATC.
- System state: The system is working correctly under normal load.
- Environment state: No relevant environmental factors.
- External stimulus: A malicious agent obtains access to the flight records database through password cracking, and downloads flight plans for commercial aircraft.
- Required system response: An audit log will be updated with login and download information to support future prosecution of malicious users.
- Response measure: The system audit contains time, IP address, and related information for the download. This information will assist in identifying and analyzing possible attacks.

Question 3

You are building a web store that you feel will unseat Amazon as the king of online shops. Your marketing department has come back with figures stating that - to accomplish your goal - your shop will need an **availability** of at least 99%, a **probability of failure on demand** of less than 0.1, and a **rate of fault occurrence** of less than 2 failures per 8-hour work period.

You have recently finished a testing period of one week (seven full 24-hour days). During this time, 972 requests were served to the page. The product failed a total of 64 times. 37 of those resulted in a system crash, while the remaining 27 resulted in incorrect shopping cart totals. When the system crashes, it takes 2 minutes to restart it.

Question 3

Want: **availability** of at least 99%, a **probability of failure on demand** of less than 0.1, and a **rate of fault occurrence** of less than 2 failures per 8-hour work period.

Currently: 972 requests. The product failed a total of 64 times (37 crashes, 27 incorrect computations). It takes 2 minutes to restart.

- What is the rate of fault occurrence?

Question 3

Want: **availability** of at least 99%, a **probability of failure on demand** of less than 0.1, and a **rate of fault occurrence** of less than 2 failures per 8-hour work period.

Currently: 972 requests. The product failed a total of 64 times (37 crashes, 27 incorrect computations). It takes 2 minutes to restart.

- What is the rate of fault occurrence?
- **64/168 hours = 0.38/hour = 3.04/8 hour work day**

Question 3

Want: **availability** of at least 99%, a **probability of failure on demand** of less than 0.1, and a **rate of fault occurrence** of less than 2 failures per 8-hour work period.

Currently: 972 requests. The product failed a total of 64 times (37 crashes, 27 incorrect computations). It takes 2 minutes to restart.

- What is the probability of failure on demand?

Question 3

Want: **availability** of at least 99%, a **probability of failure on demand** of less than 0.1, and a **rate of fault occurrence** of less than 2 failures per 8-hour work period.

Currently: 972 requests. The product failed a total of 64 times (37 crashes, 27 incorrect computations). It takes 2 minutes to restart.

- What is the probability of failure on demand?
- **64/972 = 0.066**

Question 3

Want: **availability** of at least 99%, a **probability of failure on demand** of less than 0.1, and a **rate of fault occurrence** of less than 2 failures per 8-hour work period.

Currently: 972 requests. The product failed a total of 64 times (37 crashes, 27 incorrect computations). It takes 2 minutes to restart.

- What is the availability?

Question 3

Want: **availability** of at least 99%, a **probability of failure on demand** of less than 0.1, and a **rate of fault occurrence** of less than 2 failures per 8-hour work period.

Currently: 972 requests. The product failed a total of 64 times (37 crashes, 27 incorrect computations). It takes 2 minutes to restart.

- What is the availability?
- **It was down for (37*2) = 74 minutes out of 168 hours = 74/10089 minutes = 0.7% of the time.**

Availability = 99.3%

Question 3

Want: **availability** of at least 99%, a **probability of failure on demand** of less than 0.1, and a **rate of fault occurrence** of less than 2 failures per 8-hour work period.

Currently: 972 requests. The product failed a total of 64 times (37 crashes, 27 incorrect computations). It takes 2 minutes to restart.

- Is the product ready to ship? If not, why not?

Question 3

Want: **availability** of at least 99%, a **probability of failure on demand** of less than 0.1, and a **rate of fault occurrence** of less than 2 failures per 8-hour work period.

Currently: 972 requests. The product failed a total of 64 times (37 crashes, 27 incorrect computations). It takes 2 minutes to restart.

- Is the product ready to ship? If not, why not?
- **No. Availability, POFOD are good. ROCOF is too low.**

Question 4

```
public boolean applyForVacation (String userID,
String startingDate, String endingDate)
```

- A user ID is a string in the format “firstname.lastname”, e.g., “gregory.gay”.
- The two dates are strings in the format “YYYY-DD-MM”.
- The function returns TRUE if the user was able to successfully apply for the vacation time. It returns FALSE if not. An exception can also be thrown if there is an error.

Question 4

User database with following items for each user:

- User ID
- Quantity of remaining vacation days for the user
- An array containing already-scheduled vacation dates (as starting and ending date pairs)
- An array containing dates where vacation cannot be applied for (e.g., important meetings).

Question 4

Perform category-partition testing for this function.

1. Identify choices (controllable aspects that can be varied when testing)
2. For each choice, identify representative values.
3. For each value, apply constraints (IF, ERROR, SINGLE) if they make sense.

- **Choice: Value of userID**
  - Existing user
  - Non-existing user **[error] [property not-exist]**
  - Null **[error]**
  - Malformed user ID (not in format "firstname.lastname") **[error]**
- **Choice: Value of starting date**
  - Valid date
  - Date before the current date **[error]**
  - Current date **[single]**
  - Null **[error]**
  - Malformed date (not in format "YYYY-MM-DD") **[error]**
- **Choice: Value of ending date**
  - Valid date
  - Date before the current date **[error]**
  - Current date **[single]**
  - Date before the starting date **[error]**
  - Date same as the starting date **[single]**
  - Null **[error]**
  - Malformed date (not in format "YYYY-MM-DD") **[error]**

- **Choice: Remaining vacation time for the userID**
  (Note: We are assuming the database schema prevents storing malformed/invalid values)
  - 0 days remaining
  - 1 day remaining, 1 day applied for **[single]**
  - Number of days remaining < number applied for
  - Number of days remaining = number applied for **[single]**
  - Number of days remaining > number applied for
  - User does not exist **[if not-exist]**
- **Choice: Conflicts with vacation time**
  (Note: We are assuming the database schema prevents storing malformed/invalid date ranges)
  - No conflicts with scheduled vacation or banned dates
  - Banned date(s) fall within the starting and ending dates
  - Starting date falls within already-scheduled vacation time
  - Ending date falls within already-scheduled vacation time
  - Already-scheduled vacation time falls within starting and ending dates applied for
  - The starting and ending dates fall within already-scheduled vacation time
  - User does not exist **[if not-exist]**

Question 5

Exploratory testing typically is guided by “tours”.

1. Describe one of the tours that we discussed in class.
2. Consider a banking website, where a user can do things like check their account balance, transfer funds between accounts, open new accounts, and edit their personal information. Describe three actions you might take during exploratory testing of this system, based on the tour you described above.

Question 5

- **Supermodel Tour**
  - Tests the GUI, not focused on functional correctness.
  - Visual appearance - are graphical elements in correct locations, correct size, free of rendering errors?
  - Are graphical elements/colors/fonts consistent?
  - How long does it take elements to appear?
  - Are there typos?
  - Usability issues (could this be easier to use?)
  - Accessibility issues?

Question 5

Describe three actions you might take during exploratory testing of banking system

1. Click on drop down menu - is it displayed quickly? all items present? does menu cause issues when appearing over other elements?
2. Select account - is all information displayed? is location of info correct? is info easy to find? is information readable?
3. Edit personal info - is existing info displayed? are edited segments updated and displayed correctly?

Question 6

Account
- name - personnummer - balance
Account (name, personnummer, Balance) withdraw (double amount) deposit (double amount) changeName(String name) getName() getPersonnummer() getBalance()

You are testing the Account class.

Write JUnit-format test cases to do the following:

1. Create a test case that checks a normal usage of the methods of this class.
2. Create two test cases reflecting either error-handling scenarios or quality attributes (e.g., performance or reliability).

Question 6

Account
- name - personnummer - balance
Account (name, personnummer, Balance) withdraw (double amount) deposit (double amount) changeName(String name) getName() getPersonnummer() getBalance()

- Withdraw money, verify balance.

```
@Test
public void testWithdraw_normal() {
    // Setup
    Account account = new Account("Test McTest", "19850101-1001", 48.5);
    // Test Steps
    double toWithdraw = 16.0; //Input
    account.withdraw(toWithdraw);
    double actual = account.getBalance();
    double expectedBalance = 32.5; // Oracle
    assertEquals(expected, actual); // Oracle
}
```

Question 6

Account
- name - personnummer - balance
Account (name, personnummer, Balance) withdraw (double amount) deposit (double amount) changeName(String name) getName() getPersonnummer() getBalance()

- Withdraw more than is in balance.
  - (should throw an exception with appropriate error message)

```
@Test
public void testWithdraw_moreThanBalance() {
    // Setup
    Account account = new Account("Test McTest", "19850101-1001", 48.5);
    // Test Steps
    double toWithdraw = 100.0; //Input
    Throwable exception = assertThrows(
        () -> { account.withdraw(toWithdraw); } );
    assertEquals("Amount 100.00 is greater than balance 48.50",
                 exception.getMessage()); // Oracle
}
```

Question 6

Account
- name - personnummer - balance
Account (name, personnummer, Balance) withdraw (double amount) deposit (double amount) changeName(String name) getName() getPersonnummer() getBalance()

- Withdraw a negative amount.
  - (should throw an exception with appropriate error message)

```
@Test
public void testWithdraw_negative() {
    // Setup
    Account account = new Account("Test McTest", "19850101-1001", 48.5);
    // Test Steps
    double toWithdraw = -2.5; //Input
    Throwable exception = assertThrows(
        () -> { account.withdraw(toWithdraw); } );
    assertEquals("Cannot withdraw a negative amount: -2.50",
                 exception.getMessage()); // Oracle
}
```

Let’s Take a Break

Question 7

After *carefully and thoroughly* developing a collection of requirements-based tests and running your test suite, you determine that you have achieved only 60% statement coverage. You are surprised (and saddened), since you had done a very thorough job developing the requirements-based tests and you expected the result to be closer to 100%.

Question 7

Briefly describe two (2) things that might have happened to account for the fact that 40% of the code was not exercised during the requirements-based tests.

- Few tests or poor job choosing test cases.
- Missing requirements.
- Dead or inactive code.
- Error-handling code.
- Support/interfacing code.

Question 7

Should you, in general, be able to expect 100% statement coverage through thorough requirements-based testing alone (why or why not)?

- No.
- There are almost always special cases not covered by requirements.
  - Code optimizations, support code, debug code, exception handling.

Question 7

Some structural criteria, such as MC/DC, prescribe obligations that are impossible to satisfy. What are two reasons why a test obligation may be impossible to satisfy?

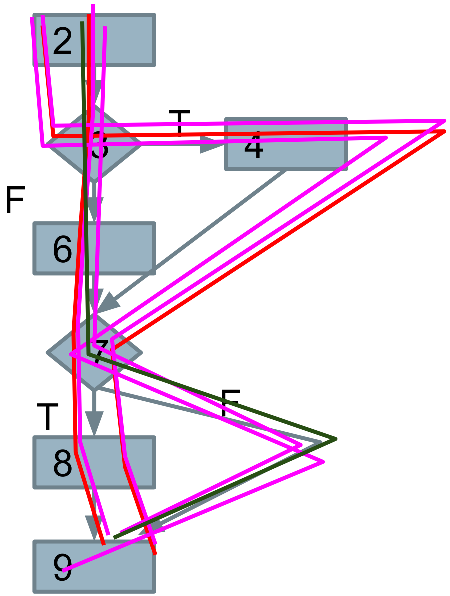
- Impossible combination of conditions
- Defensive programming (situations that may not happen in practice are planned for).
- Other situations that result in unused code (i.e., code implemented for future that is currently unreachable).

Question 8

- Draw the control-flow graph for this method.
- Develop test input that will provide statement coverage.
- Develop test input that will provide branch coverage.
- Develop test input that will provide path coverage.

```
int findMax(int a, int b, int c) {
	int temp;
	if (a > b)
		temp=a;
	else
		temp=b;
	if (c > temp)
		temp = c;
	return temp;
}
```

Question 8




```
1. int findMax(int a, int b, int c) {
2.   int temp;
3.   if (a>b)
4.         temp=a;
5.   else
6.         temp=b;
7.   if (c>temp)
8.         temp = c;
9.   return temp;
10. }
```

Statement:
(3,2,4), (2,3,4)
Branch:
(3,2,4), (3,4,1)

Path:
(4,2,5), (4,2,1), (2,3,4),
(2,3,1)

Question 8

- Modify the program to introduce a fault such that even path coverage *could* miss the fault.

Use (a > b+1) instead of (a>b) and the test input from the last slide:
(4,2,5), (4,2,1), (2,3,4), (2,3,1)
will not reveal the fault.

```
int findMax(int a, int b, int c)
{
    int temp;
    if (a>b)
        temp=a;
    else
        temp=b;
    if (c>temp)
        temp = c;
    return temp;
}
```

Question 9

- Identify all DU pairs and write test cases to achieve All DU Pair Coverage.
  - Hint - remember that there is a loop.

```
public static boolean canPartition(int[] arr) {
    Arrays.sort(arr);
    int product = 1;
    if ((Math.abs(arr[0]) >= arr[arr.length-1]) || arr[0] == 0) {
        for (int i = 1; i < arr.length; i++){
            product *= arr[i];
        }
        return arr[0] == product;
    } else{
        for (int i = 0; i < arr.length-1; i++){
            product *= arr[i];
        }
        return arr[arr.length-1] == product;
    }
}
```

Question 9

```
1.    public static boolean canPartition(int[] arr) {
2.        Arrays.sort(arr);
3.        int product = 1;
4.        if ((Math.abs(arr[0]) >= arr[arr.length-1])
      || arr[0] == 0) {
5.            for (int i = 1; i < arr.length; i++){
6.                product *= arr[i];
7.            }
8.            return arr[0] == product;
9.        } else{
10.           for (int i = 0; i < arr.length-1; i++){
11.               product *= arr[i];
12.           }
13.           return arr[arr.length-1] == product;
14.       }
15.   }
```

arr	(1, 2), (2, 4), (2, 5), (2, 6), (2, 8), (2, 10), (2, 11), (2, 13)
product	(3, 6), (6, 6), (3, 8), (6, 8), (3, 11), (11, 11), (11, 13)
i	(5, 5), (5, 6), (10, 10), (10, 11)

Question 9

```
1.   public static boolean canPartition(int[] arr) {
2.       Arrays.sort(arr);
3.       int product = 1;
4.       if ((Math.abs(arr[0]) >= arr[arr.length-1])
     || arr[0] == 0) {
5.           for (int i = 1; i < arr.length; i++){
6.               product *= arr[i];
7.           }
8.           return arr[0] == product;
9.       } else{
10.          for (int i = 0; i < arr.length-1; i++){
11.              product *= arr[i];
12.          }
13.          return arr[arr.length-1] == product;
14.      }
15.  }
```

arr	(1, 2), (2, 4), (2, 5), (2, 6), (2, 8), (2, 10), (2, 11), (2, 13)
product	(3, 6), (6, 6), (3, 8), (6, 8), (3, 11), (11, 11), (11, 13)
i	(5, 5), (5, 6), (10, 10), (10, 11)

Input	**Additional DU Pairs Covered**
`[2, 8, 4, 1]`	arr: (1, 2), (2, 4), (2, 10), (2, 11), (2, 13) product: (3, 11), (11, 11), (11, 13) i: (10, 10), (10, 11)
`[-1, -10, 0, 10]`	arr: (2, 5), (2, 6), (2, 8) product: (3, 6), (6, 6), (6, 8) i: (5, 5), (5, 6)
`[0]`	arr: (3, 8)

Question 10

Consider the following function:

```
void bSearch(int[] A, int value, int start, int end) {
    if (end <= start)
        return -1;
    mid = (start + end) / 2;
    if (A[mid] > value) {
        return bSearch(A, value, start, mid);
    } else if (value > A[mid]) {
        return bSearch(A, value, mid + 1, end);
    } else {
        return mid;
    }
}
```

1. Create an equivalent mutant.

Question 10

Consider the following function:

```
void bSearch(int[] A, int value, int start, int end) {
        if (end <= start)
                return -1;
        mid = (start + end) / 2;
        if (A[mid] > value) {
                return bSearch(A, value, start, mid);
        } else if (value > A[mid]) {
                return bSearch(A, value, mid + 1, end);
        } else {
                return mid;
        }
}
```

1. Create an equivalent mutant.

```
    } else if (value > A[mid]) {
            return bSearch(A, value,
    mid+1, end);
    } else {
    }
    return mid;
}
```

SES - End Block Shift

Question 10

Consider the following function:

```
void bSearch(int[] A, int value, int start, int end) {
    if (end <= start)
        return -1;
    mid = (start + end) / 2;
    if (A[mid] > value) {
        return bSearch(A, value, start, mid);
    } else if (value > A[mid]) {
        return bSearch(A, value, mid + 1, end);
    } else {
        return mid;
    }
}
```

2. Create an invalid mutant.

Question 10

Consider the following function:

```
void bSearch(int[] A, int value, int start, int end) {
        if (end <= start)
                return -1;
        mid = (start + end) / 2;
        if (A[mid] > value) {
                return bSearch(A, value, start, mid);
        } else if (value > A[mid]) {
                return bSearch(A, value, mid + 1, end);
        } else {
                return mid;
        }
}
```

2. Create an invalid mutant.

mid = (start + end) / 2;

~~if (A[mid] > value) {~~

```
        return bSearch(A, value, start, mid);
} else if (value > A[mid]) {
        return bSearch(A, value, mid + 1, end);
} else {
        return mid;
}
}
```

SDL - Statement Deletion

Question 10

Consider the following function:

```
void bSearch(int[] A, int value, int start, int end) {
    if (end <= start)
        return -1;
    mid = (start + end) / 2;
    if (A[mid] > value) {
        return bSearch(A, value, start, mid);
    } else if (value > A[mid]) {
        return bSearch(A, value, mid + 1, end);
    } else {
        return mid;
    }
}
```

3. Create a valid-but-not-useful mutant.

Question 10

Consider the following function:

```
void bSearch(int[] A, int value, int start, int end) {
    if (end <= start)
        return -1;
    mid = (start + end) / 2;
    if (A[mid] > value) {
        return bSearch(A, value, start, mid);
    } else if (value > A[mid]) {
        return bSearch(A, value, mid + 1, end);
    } else {
        return mid;
    }
}
```

3. Create a valid-but-not-useful mutant.

```
bSearch(A, value, start, end) {
    if (end > start)
        return -1;
    mid = (start + end) / 2;
```

ROR - Relational Operator Replacement

Question 10

Consider the following function:

```
void bSearch(int[] A, int value, int start, int end) {
        if (end <= start)
                return -1;
        mid = (start + end) / 2;
        if (A[mid] > value) {
                return bSearch(A, value, start, mid);
        } else if (value > A[mid]) {
                return bSearch(A, value, mid + 1, end);
        } else {
                return mid;
        }
}
```

3. Create a useful mutant.

```
} else if (value > A[mid]) {
        return bSearch(A, value,
mid + 2, end);
} else {
        return mid;
}

}
```

CRP - Constant for Constant Replacement

Question 11

Suppose that finite state verification of an abstract model of some software exposes a counter-example to a property that is expected to hold true for the system.

Briefly describe the follow-up actions you would take and why you would take them.

Question 11

Tells us one of the following is an issue:

- The model
  - Fault in the model, bad assumptions, incorrect interpretation of requirements
- The property
  - Property not formulated correctly.
- The requirements
  - Contradictory or incorrect requirements.

Question 12

Temporal Operators:

- **G p:** p holds globally at every state on the path from now until the end
- **F p:** p holds at some future state on the path (but not all future states)
- **X p:** p holds at the next state on the path
- **p U q:** q holds at some state on the path and p holds at every state before the first state at which q holds.
- **A:** for all paths reaching out from a state, used in CTL as a modifier for the above properties (i.e., **AG p**)
- **E:** for one or more paths reaching out from a state (but not all), used in CTL as a modifier for the above properties (i.e., **EG p**)

AG (pedestrian_light = walk -> traffic_light != green)

State variables:

- **traffic_light: {RED, YELLOW, GREEN}**
- **pedestrian_light: {WAIT, WALK, FLASH}**
- **button: {RESET, SET}**

Initially: **traffic_light = RED, pedestrian_light = WAIT, button = RESET**

Transitions:

pedestrian_light:

- **WAIT → WALK if traffic_light = RED**
- **WAIT → WAIT otherwise**
- **WALK → {WALK, FLASH}**
- **FLASH → {FLASH, WAIT}**

traffic_light:

- **RED → GREEN if button = RESET**
- **RED → RED otherwise**
- **GREEN → {GREEN, YELLOW} if button = SET**
- **GREEN → GREEN otherwise**
- **YELLOW→ {YELLOW, RED}**

button:

- **SET → RESET if pedestrian_light = WALK**
- **SET → SET otherwise**
- **RESET → {RESET, SET} if traffic_light = GREEN**
- **RESET → RESET otherwise**

G (traffic_light = RED & button = RESET -> F (traffic_light = green))

State variables:

- **traffic_light: {RED, YELLOW, GREEN}**
- **pedestrian_light: {WAIT, WALK, FLASH}**
- **button: {RESET, SET}**

Initially: **traffic_light = RED, pedestrian_light = WAIT, button = RESET**

Transitions:

pedestrian_light:

- **WAIT → WALK if traffic_light = RED**
- **WAIT → WAIT otherwise**
- **WALK → {WALK, FLASH}**
- **FLASH → {FLASH, WAIT}**

traffic_light:

- **RED → GREEN if button = RESET**
- **RED → RED otherwise**
- **GREEN → {GREEN, YELLOW} if button = SET**
- **GREEN → GREEN otherwise**
- **YELLOW→ {YELLOW, RED}**

button:

- **SET → RESET if pedestrian_light = WALK**
- **SET → SET otherwise**
- **RESET → {RESET, SET} if traffic_light = GREEN**
- **RESET → RESET otherwise**

Negate to get trap property:
G !(button = SET -> F (pedestrian_light = WALK))

State variables:

- **traffic_light: {RED, YELLOW, GREEN}**
- **pedestrian_light: {WAIT, WALK, FLASH}**
- **button: {RESET, SET}**

Initially: **traffic_light = RED, pedestrian_light = WAIT, button = RESET**

Transitions:

pedestrian_light:

- **WAIT → WALK if traffic_light = RED**
- **WAIT → WAIT otherwise**
- **WALK → {WALK, FLASH}**
- **FLASH → {FLASH, WAIT}**

traffic_light:

- **RED → GREEN if button = RESET**
- **RED → RED otherwise**
- **GREEN → {GREEN, YELLOW} if button = SET**
- **GREEN → GREEN otherwise**
- **YELLOW→ {YELLOW, RED}**

button:

- **SET → RESET if pedestrian_light = WALK**
- **SET → SET otherwise**
- **RESET → {RESET, SET} if traffic_light = GREEN**
- **RESET → RESET otherwise**

Question 13

Microwave controller

- Door: {Open, Closed} -- sensor input indicating state of the door
- Button: {None, Start, Stop} -- button press
- Timer: 0...999 -- (remaining) seconds to cook
- Cooking: Boolean -- state of the heating element

In CTL:

- The microwave shall never cook when the door is open.
- **AG (Door = Open -> !Cooking)**

Question 13

Microwave controller

- Door: {Open, Closed} -- sensor input indicating state of the door
- Button: {None, Start, Stop} -- button press
- Timer: 0...999 -- (remaining) seconds to cook
- Cooking: Boolean -- state of the heating element

In CTL:

- The microwave shall cook only as long as there is remaining cook time.
- **AG (Cooking -> Timer > 0)**

Question 13

Microwave controller

- Door: {Open, Closed} -- sensor input indicating state of the door
- Button: {None, Start, Stop} -- button press
- Timer: 0...999 -- (remaining) seconds to cook
- Cooking: Boolean -- state of the heating element

In CTL:

- If the stop button is pressed when the microwave is not cooking, the remaining cook time shall be cleared.
- **AG (Button = Stop & !Cooking -> AX (Timer = 0))**

Question 13

Microwave controller

- Door: {Open, Closed} -- sensor input indicating state of the door
- Button: {None, Start, Stop} -- button press
- Timer: 0...999 -- (remaining) seconds to cook
- Cooking: Boolean -- state of the heating element

In LTL:

- It shall never be the case that the microwave can continue cooking indefinitely.
- **G (Cooking -> F (!Cooking))**

Question 13

Microwave controller

- Door: {Open, Closed} -- sensor input indicating state of the door
- Button: {None, Start, Stop} -- button press
- Timer: 0...999 -- (remaining) seconds to cook
- Cooking: Boolean -- state of the heating element

In LTL:

- The only way to initiate cooking shall be pressing the start button when the door is closed and the remaining cook time is not zero.
- **G (!Cooking U ((Button = Start & Door = Closed) & (Timer > 0)))**

Question 13

Microwave controller

- Door: {Open, Closed} -- sensor input indicating state of the door
- Button: {None, Start, Stop} -- button press
- Timer: 0...999 -- (remaining) seconds to cook
- Cooking: Boolean -- state of the heating element

In LTL:

- The microwave shall continue cooking when there is remaining cook time unless the stop button is pressed or the door is opened.
- **G ((Cooking & Timer > 0) -> X (((Cooking | (!Cooking & Button = Stop)) | (!Cooking & Door = Open)))**

Any other questions?

Thank you for being a great class!

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY